FilePack

Introduction

As the name implies, FilePack is a collection of external routines that provide the 4th DIMENSION developer with access to many useful and otherwise inaccessible features of the file storage subsystem of the MacOS and Windows operating systems. This release of FilePack provides support for the new "plug-in" file format for extensions used by 4D v3.5, as well as native code support for the PowerPC processor and the WIN32 operating systems used by Intel computers (Windows NT, Windows 95 and Windows 3.x with WIN32s).

Before we get to the details of how to use FilePack, we need to go through the routine legal stuff:

You may distribute FilePack only under the following conditions:

1. You may distribute the unaltered self-extracting archive file, which includes this documentation and the FilePack software files in their complete, original and unedited forms. No fee of any kind can be associated with such distribution. This means no fee for expenses, consultation, installation or whatever. If you want an exception to this, contact me.
2. You may distribute the external package as part of a commercial, shareware or contract database application, so long as no fee is charged for the package itself. If the receiver has access to the source code, you must give them a copy of this documentation file, unaltered, as well. If you receive any money for the distribution, do NOT give the client the impression that they are paying for FilePack and for the right to call me for support.
3. You may NOT sell FilePack by itself, or as part of any collection or library of externals, or as part of any 4D development "toolbox" or "shell". If you have questions or are uncertain regarding your right to use FilePack with a specific product, please call and we can talk it over.

FilePack is being released under this license for those of you who are already using a previous release of FilePack, and who are satisfied with its status as a free tool. FilePack is also available as part of my commercial product, PowerPacks. PowerPacks v2.5 has over 300 routines in 17 separate packages, and is compatible with both 4th DIMENSION and 4D Server on the Macintosh and Windows. Please contact me if you have any questions.

I am eager to hear suggestions, constructive criticism and bug reports regarding FilePack. I can be contacted at:

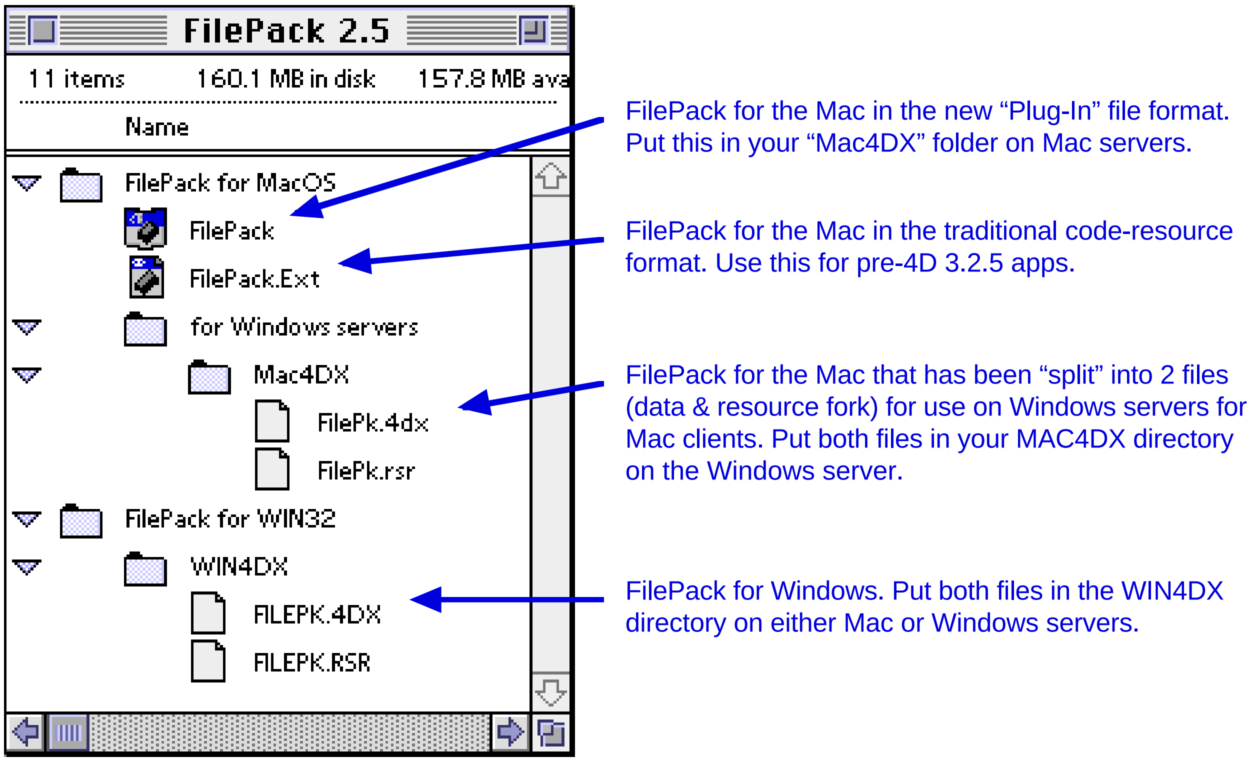
Bob Pulgino, RKP Software
8533 Crestview Drive
Fairfax, VA 22031-2802

E-mail:	bpulgino@clark.net (until July 1, 1996)
	bob@rkp.com (thereafter)
Voice:	(703) 205-9845
Fax:	(703) 205-9846

The previous version of FilePack was compiled into four separate packages, because of the memory requirements that the old format for external packages (code resources) made on your applications. With code resources, the size of each package was generally more significant than the number of packages used by an application when determining the impact that extensions would have on its memory requirements.

With the advent of the new "plug-in" file format, the number of individual packages used by a given application has more of an impact on the memory requirements than the size of the given packages. As a result, we have reversed the earlier trend and recompiled FilePack as a single package. Because of this change, it is important that you manually delete all old copies of the FilePack packages before installing this new package if you are updating from a previous release of FilePack.

This release of FilePack is distributed in four different formats.:



FilePack for the Mac in the new "Plug-In" file format. Put this in your "Mac4DX" folder on Mac servers.

FilePack for the Mac in the traditional code-resource format. Use this for pre-4D 3.2.5 apps.

FilePack for the Mac that has been "split" into 2 files (data & resource fork) for use on Windows servers for Mac clients. Put both files in your MAC4DX directory on the Windows server.

FilePack for Windows. Put both files in the WIN4DX directory on either Mac or Windows servers.

Because of the new file formats, OS platforms, and the myriad of different configurations of the 4D tools you can use to deploy your applications, making use of extensions like FilePack isn't always as easy as it used to be. Be sure to carefully follow the instructions given by your particular 4th DIMENSION interpreter and compiler.

File System Attributes

A computer's operating system maintains a great deal of information on each disk about the files stored there, in addition to just the file names and content. When your application has to work with existing files or create new ones, you often will need to be able to access and change this information as well.

The Macintosh OS maintains the following attributes for each file:

Type Code: A four-character code that is intended to be used by applications to identify the type of information contained in the file and its organization or structure. Commonly used file types include "TEXT" for simple ASCII text files, "PICT" for generic graphic files and "APPL" for application files. Files created by commercial software with proprietary contents have unique type codes as well; for example, 4th Dimension assigns a type code of "BAS5" to its database-structure documents.

Creator Code: This is also a four-character code, used by the Finder to link documents with the application that created it. When you ask the Finder to open a document, it will first look at the creator-code for that document. It will then search for an application with the same creator code, and if it finds one, it will use that application to open the document.

Locked Flag: If the locked flag is set, no application (including your application and the Finder) will be able rename, trash or modify the file or folder, without unlocking it first. The user will be able to unlock it by un-checking the Locked checkbox in the Finder's Get Info dialog.

Invisible Flag: If this flag is set, the file or folder will not be visible to the user in any of the Finder's direc-tory windows. The use of invisible files is discouraged these days, and should only be considered for special circumstances; an invisible file is impossible to detect or manipulate in any way by the typical user.

Stationery Flag: If this flag is set for a document, the document will be treated in a special way by the Finder and its creator application when it is opened: instead of editing the document itself, the user will be given an untitled copy of the document, and when the copy is saved, the original stationery document will remain intact. The goal of this mechanism is to allow documents to be created that will be used as templates, starting points for new documents intended for a specific, user-defined purpose. A stationery document will be given a unique icon that resembles a pad of paper.

Creation Date & Time: The Finder keeps track of the exact date and time of day that each file and folder was created. This information could be useful to you in determining the age or relevance of the infor-mation contained in a given file.

Modification Date & Time: The Finder also keeps track of the exact date and time of day that each file and folder was most recently modified. This information could be useful to you when comparing files, to determine which file contains the most recent information.

Backup Date & Time: These fields are meant to be used by specialized backup utilities, to keep track of when a file or folder was last backed up. They are not maintained by the Finder. If you make use of them, be careful not to end up interfering with your users' backup methods.

Logical Size: The logical size of a file is the measure of the exact number of bytes of information it contains (counting both the data and resource forks).

Physical Size: The physical size of a file is the measure of the amount of space allocated to store it on the disk. Space is allocated to a file in discrete, constant-size blocks; the size of the allocation block de-pends on the overall size of the volume, but is usually 1,024 bytes.

On the Windows platform, some of these attributes are available and some aren’t. A summary of the differences between the file-system attributes used by the MacOS and the WIN32 API follows:

- WIN32 does not support the use of file type- and creator-code attributes. Instead, a predefined file name “extension” of one to three characters, separated from the rest of the file name by a period, is used to define both the type of data contained in the file and which application should be used to open it. 4th Dimension for Windows makes use of an internal table to map standard MacOS file type codes to common Windows file extensions, and PowerPacks supports this mechanism. You can extend this table using the new MAP FILE TYPES command.
- WIN32 refers to the “Locked” flag as the “Read-Only” attribute, and the “Invisible” flag as the “Hidden” attribute.
- WIN32 has no attribute that is analogous to the MacOS “Stationery” flag.
- WIN32 has no direct analog to the MacOS “Backup Date & Time”; it instead uses the “Archive” boolean attribute, which a future version of PowerPacks will enable you to access.

GetFileCreator

Description:

GetFileCreator*(FilePath:T): CreatorCode:S*

This function enables you to determine the creator code of a file.

Parameters:

FilePath.............................. A text expression containing the fully-qualified path to the file.

Return Value:

CreatorCode .... The 4-byte creator code for the file.

Platform Notes:

On the Windows platform, this routine always returns “????”, since there is no file attribute which corresponds to the Macintosh creator code.

GetFileType

Description:

GetFileType*(FilePath:T): TypeCode:S*

This function enables you to determine the file-type code of a specific file.

Parameters:

FilePath.............................. A text expression containing the fully-qualified path to the file.

Return Value:

TypeCode.......................... The 4-byte type code for the file.

Platform Notes:

Under Windows, this routine uses 4D’s internal table to determine the type code which corresponds to the extension of the specified file. You can extend this table to include the type codes and extensions of your files using the MAP FILE TYPES command.

SetFileCreator

Description:

SetFileCreator*(FilePath:T; CreatorCode:S): ResultCode:I*

This function enables you to assign a new creator code to a file.

Parameters:

FilePath	A text expression containing the fully-qualified path to the file.
CreatorCode	The new 4-byte creator code for the file.

Return Value:

ResultCode	A code indicating if the creator code was assigned successfully. A non-zero result indicates that an error occurred.

Platform Notes:

On the Windows platform, this routine does nothing since there is no file attribute which corresponds to the Macintosh creator code.

SetFileType

Description:

SetFileType*(FilePath:T; TypeCode:S): ResultCode:I*

This function enables you to assign a new file-type code to a file.

Parameters:

FilePath	A text expression containing the fully-qualified path to the file.
TypeCode	The new 4-byte file-type code for the file.

Return Value:

ResultCode	An error code indicating whether or not the code was assigned successfully. A non-zero result indicates that there was an error.

Platform Notes:

Under Windows, this routine uses 4D’s internal table to determine the file extension which corresponds to the specified type code, and if found, it changes the file extension accordingly. You can extend this table to include the type codes and extensions of your files using the MAP FILE TYPES command.

GetFileInfo

Description:

GetFileInfo*(FilePath:T; CreatorCode:S; TypeCode:S): ResultCode:I*

This function enables you to read both the creator and file-type codes of a file in a single step.

Parameters:

FilePath	A text expression containing the fully-qualified path to the file.
CreatorCode	Returns the 4-byte creator code for the file. Since this parameter returns a value, you must use a global variable of type STRING (4 bytes or larger) for this parameter.
TypeCode	Returns the 4-byte type code for the file. Since this parameter returns a value, you must use a global variable of type STRING (4 bytes or larger) for this parameter.

Return Value:

ResultCode	An error code indicating whether or not the codes were read successfully. A non-zero result indicates that there was an error.

Platform Notes:

Under Windows, the creator code returned by this routine will always be "????", since there is no corresponding file attribute. The type code returned will be based on the extension of the specified file, as defined by 4D's internal MAP FILE TYPES table.

SetFileInfo

Description:

SetFileInfo*(FilePath:T; CreatorCode:S; TypeCode:S): ResultCode:I*

This function enables you to assign a new file-type and creator code for a file in a single step.

Parameters:

FilePath	A text expression containing the fully-qualified path to the file.
CreatorCode	A four-byte string containing the file-creator code to be assigned to the file.
TypeCode	A four-byte string containing the file-type code to be assigned to the file.

Return Value:

ResultCode	An error code indicating whether or not the file codes were assigned successfully. A non-zero result indicates that there was an error.

Platform Notes:

Under Windows, the creator code passed when using this routine is ignored, since there is no corresponding file attribute. The routine uses 4D's internal table to determine the file extension which corresponds to the specified type code, and if found, it changes the file extension accordingly. You can extend this table to include the type codes and extensions of your files using the MAP FILE TYPES command.

GetFileCDate

Description:

GetFileCDate*(Path:T): CreatedDate:D*

This function enables you to determine the creation date of a file or folder.

Parameters:

Path	A text expression containing the fully-qualified path to the file or folder.

Return Value:

CreatedDate	The date recorded by the Finder that the file or folder was created.

GetFileCTime

Description:

GetFileCTime*(Path:T): CreatedTime:L*

This function enables you to determine the time of day that a file or folder was created.

Parameters:

Path	A text expression containing the fully-qualified path to the file or folder.

Return Value:

CreatedTime	The time of day recorded by the Finder that the file or folder was created. The time is returned as a long integer representing the number of seconds since midnight.

GetFileCSeconds

Description:

GetFileCSeconds*(Path:T): CreatedSeconds:L*

This function returns the date and time that a specified file or folder was created, expressed as a single value, the total number of seconds. Using this value instead of the date and time returned by GetFileCDate and GetFileCTime allows you to make comparisons more easily.

Parameters:

Path	A text expression containing the fully-qualified path to the file or folder.

Return Value:

CreatedSeconds	The date and time that the specified file or folder was created, expressed as the number of seconds since midnight, January 1, 1904.

Platform Notes:

Under the Macintosh OS, the value returned represents the number of seconds since midnight, January 1, 1904. Under Windows, it represents the number of seconds since midnight, January 1, 1601.

SetCDateTime

Description:

SetCDateTime(*Path:T; CreatedDate:D; CreatedTime:L*): *Result:I*

This function enables you to assign a new created date and time to a file or folder.

Parameters:

Path	A text expression containing the fully-qualified path to the file or folder.
CreatedDate	A date expression providing the new creation date.
CreatedTime	A long-integer expression providing the new creation time.

Return Value:

Result	An error code indicating whether or not the date and time were assigned successfully. A non-zero result indicates that there was an error.

GetFileMDate

Description:

GetFileMDate(*Path:T*): *ModDate:D*

This function enables you to determine the date a file or folder was last modified.

Parameters:

Path	A text expression containing the fully-qualified path to the file or folder.

Return Value:

ModDate	The date recorded by the Finder that the file or folder was last modified.

GetFileMTime

Description:

GetFileMTime(*Path:T*): *ModTime:L*

This function enables you to determine the time of day that a file or folder was last modified.

Parameters:

Path	A text expression containing the fully-qualified path to the file or folder.

Return Value:

ModTime	The time of day recorded by the Finder that the file or folder was last modified. The time is returned as a long integer representing the number of seconds since midnight.

GetFileMSeconds

Description:

GetFileMSeconds*(Path:T): ModSeconds:L*

This function returns the date and time that a specified file or folder was last modified, expressed as a single value, the total number of seconds. Using this value instead of the date and time returned by GetFileMDate and GetFileMTime allows you to make comparisons of files more easily.

Parameters:

Path	A text expression containing the fully-qualified path to the file or folder.

Return Value:

ModSeconds	The date and time that the specified file or folder was most recently modified, expressed as a single value, the number of seconds since midnight, January 1, 1904.

Platform Notes:

Under the Macintosh OS, the value returned represents the number of seconds since midnight, January 1, 1904. Under Windows, it represents the number of seconds since midnight, January 1, 1601.

SetMDateTime

Description:

SetMDateTime*(Path:T; ModDate:D; ModTime:L): Result:I*

This function enables you to assign a new modified date and time to a file or folder.

Parameters:

Path	A text expression containing the fully-qualified path to the file or folder.
ModDate	A date expression providing the new modification date.
ModTime	A long-integer expression providing the new modification time.

Return Value:

Result	An error code indicating whether or not the date and time were assigned successfully. A non-zero result indicates that there was an error.

GetFileBDate

Description:

***GetFileBDate**(Path:T): BackedUpDate:D*

This function enables you to determine the date a file or folder was last backed up. Note that the Finder does not automatically maintain the backed-up date and time attributes; unless specialized software is used, this value is likely to be zero (!01/01/04!).

Parameters:

Path	A text expression containing the fully-qualified path to the file or folder.

Return Value:

BackedUpDate	The date recorded by the user's backup utility that the file or folder was last backed up.

Platform Notes:

The Windows platform has no corresponding file attribute, so this routine always returns a zero date value (!01/01/04!).

GetFileBTime

Description:

***GetFileBTime**(Path:T): BackedUpTime:L*

This function enables you to determine the time of day that a file or folder was last backed up. Note that the Finder does not automatically maintain the backed-up date and time attributes; unless specialized software is used, this value is likely to be zero (†00:00:00†).

Parameters:

Path	A text expression containing the fully-qualified path to the file or folder.

Return Value:

BackedUpTime	The time of day recorded by the user's backup utility that the file or folder was last modified. The time is returned as a long integer representing the number of seconds since midnight.

Platform Notes:

The Windows platform has no corresponding file attribute, so this routine always returns a zero time value (†00:00:00†).

GetFileBSeconds

Description:

***GetFileBSeconds**(Path:T): BackedUpSeconds:L*

This function returns the date and time that a specified file or folder was last backed up, expressed as a single value, the number of seconds since midnight, January 1, 1904. Using this value instead of the date and time returned by GetFileBDate and GetFileBTime allows you to make comparisons of files more easily. Note that the Finder does not automatically maintain the backed-up date and time attributes; unless specialized software is used, these values are likely to be zero (!01/01/04! and †00:00:00†).

Parameters:

Path	A text expression containing the fully-qualified path to the file or folder.

Return Value:

BackedUpSeconds	The date and time that the specified file or folder was most recently backed, expressed as a single value, the number of seconds since midnight, January 1, 1904.

Platform Notes:

The Windows platform has no corresponding file attribute, so this routine always returns zero values.

SetBDateTime

Description:

***SetBDateTime**(Path:T; BackedUpDate:D; BackedUpTime:L): Result:I*

This function enables you to assign a new backed-up date and time to a file or folder. Note that many specialized backup utilities depend on this date to properly perform incremental backups of the files on a hard disk. If you use this routine indescriminantly, you run the risk of interfering with your users' backup procedures.

Parameters:

Path	A text expression containing the fully-qualified path to the file or folder.
BackedUpDate	A date expression providing the new backed-up date.
BackedUpTime	A long-integer expression providing the new backed-up time.

Return Value:

Result	An error code indicating whether or not the date and time were assigned successfully. A non-zero result indicates that there was an error.

Platform Notes:

This routine has no effect when used under Windows, since that platform has no corresponding file attribute.

GetFileLogSize

Description:

***GetFileLogSize**(FilePath:T): LogSize:L*

This function returns the logical size of a file; i.e. the number of bytes that could actually be imported from the file.

Parameters:

FilePath	A text expression containing the fully-qualified file name.

Return Value:

LogSize	The number of bytes of data in the file.

GetFilePhySize

Description:

***GetFilePhySize**(FilePath:T): PhySize:L*

This function determines the physical size of a file; i.e. the number of bytes of space it occupies on disk.

Parameters:

FilePath	A text expression containing the fully-qualified file name.

Return Value:

PhySize	The number of bytes of space occupied by the file on disk.

GetFileLocked

Description:

***GetFileLocked**(FilePath:T): ResultCode:I*

This function enables you to determine if the specified file has been locked.

Parameters:

FilePath	A text expression containing the fully-qualified file name.

Return Value:

ResultCode	A zero if the file is unlocked, a one if the file is locked, or a negative number if an error occurred.

SetFileLocked

Description:

SetFileLocked*(FilePath:T; SetFlag:I): ResultCode:I*

This function enables you to lock or unlock the specified file.

Parameters:

FilePath	A text expression containing the fully-qualified file name.
SetFlag	A number specifying whether the file is to be locked or unlocked. If this value is zero, the file will be unlocked; if it is 1 (or any non-zero value), the file will be locked.

Return Value:

ResultCode	An error code indicating whether or not the file was locked or unlocked successfully. A non-zero result indicates that there was an error.

GetFileStatnry

Description:

GetFileStatnry*(FilePath:T): ResultCode:I*

This function enables you to determine if the specified file has been marked as a “stationery” document, either by the user via the Finder’s Get Info dialog, or by the application that created it.

Parameters:

FilePath	A text expression containing the fully-qualified file name.

Return Value:

ResultCode	A zero if the file is not a stationery pad, a one if it is, or a negative number if an error occurred.

Platform Notes:

This routine always returns zero when used under Windows, since that platform has no corresponding file attribute.

SetFileStatnry

Description:

SetFileStatnry*(FilePath:T; SetFlag:I): ResultCode:I*

This function enables you to set or reset the attribute of a file that makes it a Finder “stationery” document.

Parameters:

FilePath	A text expression containing the fully-qualified file name.
SetFlag	A number specifying whether the file is to be a stationery document or not. If this value is a 1 (or any non-zero value), the file will be made into a stationery document (the stationery attribute will be set); if it is a zero, the stationery attribute will be reset and the file will not be a stationery document.

Return Value:

ResultCode	An error code indicating whether or not the file's attribute was set successfully. A non-zero result indicates that there was an error.

Platform Notes:

This routine has no effect when used under Windows, since that platform has no corresponding file attribute.

GetFileVisible

Description:

GetFileVisible*(FilePath:T): ResultCode:I*

This function enables you to determine if the specified file has been marked as a "hidden" or "invisible" file, such that it will not appear in its Finder window or in a standard file dialog.

Parameters:

FilePath	A text expression containing the fully-qualified file name.

Return Value:

ResultCode	If the file is visible (has not been made hidden), a value of 1 is returned; a zero value is returned if the file is invisible (hidden from the user in the Finder and standard-file dialogs). A negative error code is returned if an error occurs.

SetFileVisible

Description:

SetFileVisible*(FilePath:T; SetFlag:I): ResultCode:I*

This function enables you to set or reset the "invisible" attribute of the specified file.

Parameters:

FilePath	A text expression containing the fully-qualified file name.
SetFlag	A number specifying whether the file is to be made visible or not. If this value is a 1, the file will be made visible to the user; if it is a zero, the invisible attribute will be set and the file will not be visible to the user.

Return Value:

ResultCode	An error code indicating whether or not the file's attribute was set successfully. A non-zero result indicates that there was an error.

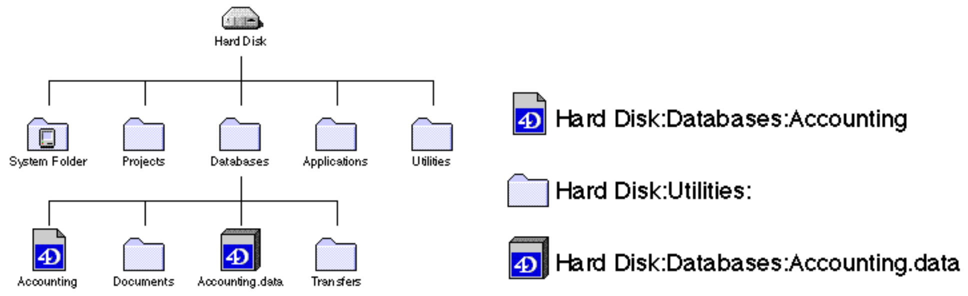
The Hierarchical Filing System

The principle job of the OS file system is to organize files on a disk into user-defined groups called folders or directories. Each folder can contain files and other folders, which in turn can contain more files and folders, and so on. On the Macintosh, this hierarchical arrangement of files and folders has been appropriately dubbed the Hierarchical File System (HFS).

In order to perform an operation on a document or folder with a PowerPacks routine, you need to completely identify it by supplying its full "path" name. The path name of a file or folder always begins with the name of the volume it is stored on, and ends with the name of the object itself. If the object is within a folder, the name of that folder must be included in the path, between the volume and object name; if the folder is "nested" within one or more other folders, the names of each folder must be included in the path name, in the top-to-bottom order of the folder hierarchy. Finally, each name component in the path must be separated by a colon character; if the object itself is a folder, the path should end with a colon character as well.

Each file entry on a Macintosh volume actually represents two physical components, a "data fork" and a "resource fork". The resource fork is used to store static data structures used by Macintosh applications for a wide variety of purposes.

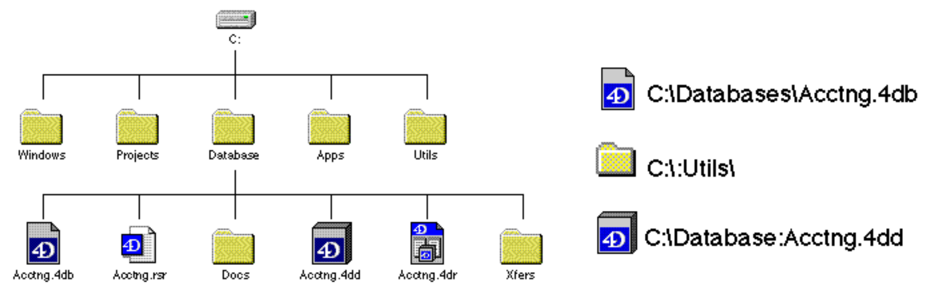
In the illustration below, an example of a typical hierarchical arrangement of files and folders is depicted, along with a few examples of the path names for items in that arrangement.



On a Windows machine, these concepts are basically the same, with the following minor exceptions:

- Windows volumes are not given a name by the user in the same way as Macintosh volumes. Windows volumes are referred to by a single letter which designates a physical or logical disk drive, followed by a colon character. Diskette drive volumess are usually referred to as "A:' and "B:", "C:" usually refers to the startup hard disk volume, and "D:" through "Z:" are used to refer to secondary hard disks, CR-ROMs and network volumes.
- The component names that make up a path name are delimited using the back-slash character "\", instead of a colon.
- The MS-DOS operating system, upon which Windows is based, limits names of files and folders to 8 characters and total path name length to 260 characters.(Windows NT and Windows 95 support new files systems that allow users to specify names longer than this.)
- Windows file systems do not support the concept of separate data and resource forks for a given file; 4th Dimension works around this difference by storing the resource fork in a separate file which has the same name as the data file, but uses a special filename extension (usually ".rsr").

The illustration below shows how the previous example would look on a Windows machine:



Each folder on a given volume is actually a separate catalog containing the locations and attributes of each file and folder it contains. The top (or "root") level of each volume is also a separate catalog for those objects not nested within other folders. When we refer to a file-system catalog, therefore, we are referring to the list of files and folders contained within a specific folder or at the root level of a volume. FilePack provides several routines for basic operations with HFS catalogs:

- The NewFolder routine enables you to create a new folder anywhere on a given volume.
- The HFSParentName and HFSShortName routines are provided to give you an easy way to break full path names down into their component file and folder names.
- The CopyFile routine allows you to make a copy of a file to any folder on any mounted volume.
- The HFSCopy routine allows you to copy entire directories.
- The HFSMove routine enables you to move a file or folder from one catalog to another on a given volume.
- The HFSRename routine enables you to rename a file, folder or volume.
- The HFSDelete routine will delete a file or a folder (if it is empty).
- The HFSExists routine allows you to test to see if an object with a given path name is already defined.
- The HFSCatToArray routine is used to load the contents of a specific catalog into an array.
- HFSGetCatCount and HFSGetCatItem allow you to "step through" the items in a catalog, one at a time.

NewFolder

Description:

***NewFolder**(NewFolderPath:T): ResultCode:I*

Given a pathname of a non-existing folder whose parent path does exist (such as returned by PutFileName), this function creates the folder.

Parameters:

NewFolderPath .... The pathname of the new folder. The pathname must specify a new name in an existing directory.

Return Value:

ResultCode .... Zero if the folder was created successfully; if an error occurs, the error code will be returned.

HFSParentName

Description:

HFSParentName*(ThePath:T): ParentPath:T*

This function returns the path for the folder that a given file or folder is in; it simply strips away the file's simple name from its fully-qualified name.

Parameters:

ThePath	A text expression containing the fully-qualified file or folder path.

Return Value:

ParentPath	The path for the specified file name; i.e., all characters in the parameter value up to and including the last colon in the text value.

HFSShortName

Description:

HFSShortName*(ThePath:T): ShortName:T*

This function returns the simple name for a given file or folder; it simply strips away the object's path name from its fully-qualified name. This performs the complimentary function to the HFSParentName routine.

Parameters:

ThePath	A text expression containing the fully-qualified file or folder path.

Return Value:

ShortName	The simple name for the specified file or folder; i.e., all characters in the parameter value following the last path delimiter in the string.

CopyFile

Description:

CopyFile*(OldFilePath:T; NewFilePath:T): ResultCode:I*

This function allows you to make a copy of an existing file to any folder on any mounted disk. The copy will have the same file-system attributes as the original.

Parameters:

OldFilePath	A text expression containing the fully-qualified name of the file to be copied.
NewFilePath	A text expression containing the fully-qualified name of the copy of the file to be created. The NewFilePath parameter specifies which disk will hold the copy, the folder to put it in, and what its name should be. Therefore, this path must refer to a non-existing file in an existing directory; i.e. the volume name and folders specified must already exist, but the file name must not.

Return Value:

ResultCode	An error code indicating whether or not the file copy was created successfully. A non-zero result indicates that there was an error.

Platform Notes:

On the Macintosh platform, both the data and resource forks will be copied. On Windows, the routine will look for an associated resource file and attempt to copy it as well.

If you want to include the resource fork when running on the Mac, but ignore the resource file when running on Windows, copy the files individually using the HFSCopyData routine.

If you want to ignore the resource forks on both the Mac and Windows, use the CopyDataFile routine.

CopyDataFile

Description:

***CopyDataFile**(OldFilePath:T; NewFilePath:T): ResultCode:I*

This function works almost the same as the CopyFile routine, with the exception that it ignores the resource fork or RSR file associated with the file to be copied. The resulting copy will have the same file attributes and data fork as the original, but it will have no resource fork.

Parameters:

OldFilePath	A text expression containing the fully-qualified name of the file to be copied.
NewFilePath	A text expression containing the fully-qualified name of the copy of the file to be created. The NewFilePath parameter specifies which disk will hold the copy, the folder to put it in, and what its name should be. Therefore, this path must refer to a non-existing file in an existing directory; i.e. the volume name and folders specified must already exist, but the file name must not.

Return Value:

ResultCode	An error code indicating whether or not the file copy was created successfully. A non-zero result indicates that there was an error.

Platform Notes:

This routine is useful on each platform under different circumstances. On the Macintosh, this routine allows you to copy data files and explicitly ignore the resources forks of these source files, either because they are not needed or because the source files are located on foriegn file servers that do not support resource forks.

Many foriegn servers have a bug which, when standard Macintosh File Manager routines are used to query for the size of a given file's resource fork, cause them to report arbitrary values. In these cases, the 4D developer knows that the file to be copied has no resource fork anyway. By using the CopyDataFile routine, the developer can avoid potential problems such as copies of files with randomly-sized resource forks full of garbage and system crashes.

On the Windows platform, the CopyFile routine will automatically look for a file with the same name and path as the specified source file, but with an extension of "RSR". If such a file is found, the routine will assume that it is the converted resource fork for the source file and will copy it as well. In situations where this is inappropriate, you can use the CopyDataFile routine instead, which will ignore the RSR files.

If you need a routine which will copy both forks when used on the Macintosh, but will ignore the RSR file when used on Windows, use the HFSCopyData routine and copy each file individually.

HFSCopy

Description:

HFSCopy*(SourcePath:T; DestPath:T): ResultCode:I*

This routine is an enhanced version of the CopyFile routine. It will not only copy individual files, but can also copy entire folders with their contents intact. It will offer improved performance when copying files on remote volumes from the same file server.

Parameters:

SourcePath	A text expression containing the fully-qualified name of the file or folder to be copied.
DestPath	A text expression containing the fully-qualified path to the folder that the source is to be copied into. The duplicate of the source will be created in this specifed folder, with the same simple name as the source. If an object with this name already exists in the destination folder, an error will be returned.

Return Value:

ResultCode	An error code indicating whether or not the copy was created successfully. A non-zero result indicates that there was an error.

Platform Notes:

When used on the Macintosh platform to copy an individual file, both the data and resource forks will be copied. On Windows, the routine will look for an associated resource file and attempt to copy it as well.

When used to copy a folder, the entire content of the folder will always be copied.

HFSCopyData

Description:

HFSCopyData*(SourcePath:T; DestPath:T): ResultCode:I*

On the Macintosh platform, this routine always works the same as the HFSCopy routine: it allows you to copy individual files or entire folders from one place to another. Both the resource and data forks will always be copied.

On the Windows platform, this routine has the same purpose but behaves differently than HFSCopy in one way: when the source path points to an individual file, HFSCopy will automatically look for a file with the same path and name, but with an “RSR” extension. If found, HFSCopy will assume that this file contains the converted Macintosh resource fork for the file, and will attempt to copy it as well. HFSCopyData, on the other hand, will ignore the RSR files.

In all cases when the source path points to a folder, every file in the folder will be copied (including RSR files).

Parameters:

SourcePath	A text expression containing the fully-qualified name of the file or folder to be copied.
DestPath	A text expression containing the fully-qualified path to the folder that the source is to be copied into. The duplicate of the source will be created in this specifed folder, with the same simple name as the source. If an object with this name already exists in the destination folder, an error will be returned.

Return Value:

ResultCode	An error code indicating whether or not the copy was created successfully. A non-zero result indicates that there was an error.

HFSMove

Description:

HFSMove*(CurrentPath:T; NewParentPath:T): ResultCode:I*

Given a pathname of a file or folder in the first parameter, and the pathname to any other folder on the same volume in the second parameter, will move the existing file or folder to the subdirectory of the other folder.

Parameters:

CurrentPath	The path to the file or folder that you wish to move to a different location in the volume's folder hierarchy.
NewParentPath	The path to the folder that is to contain the existing file or folder after the move. This folder must be on the same volume as the file or folder to be moved.

Return Value:

ResultCode	Zero if the move was completed successfully; if an error occurs, the error code will be returned.

HFSMoveData

Description:

HFSMoveData*(CurrentPath:T; NewParentPath:T): ResultCode:I*

Given a pathname of a file or folder in the first parameter, and the pathname to any other folder on the same volume in the second parameter, will move the existing file or folder to the subdirectory of the other folder.

Parameters:

CurrentPath	The path to the file or folder that you wish to move to a different location in the volume's folder hierarchy.
NewParentPath	The path to the folder that is to contain the existing file or folder after the move. This folder must be on the same volume as the file or folder to be moved.

Return Value:

ResultCode	Zero if the move was completed successfully; if an error occurs, the error code will be returned.

HFSRename

Description:

***HFSRename**(CurrentPath:T; NewName:S): ResultCode:I*

The function enables you to change the name of a file, folder or volume.

Parameters:

CurrentPath	The path to the file or folder that you wish to rename.
NewName	The new simple name (do not re-specify the entire path) for the file or folder.

Return Value:

ResultCode	Zero if the file or folder was renamed successfully; if an error occurs, the error code will be returned. Usually, such errors will be caused by the specification of a non-existing file or folder for CurrentPath, or of an existing name for NewName.

HFSRenameData

Description:

***HFSRenameData**(CurrentPath:T; NewName:S): ResultCode:I*

The function enables you to change the name of a file, folder or volume.

Parameters:

CurrentPath	The path to the file or folder that you wish to rename.
NewName	The new simple name (do not re-specify the entire path) for the file or folder.

Return Value:

ResultCode	Zero if the file or folder was renamed successfully; if an error occurs, the error code will be returned. Usually, such errors will be caused by the specification of a non-existing file or folder for CurrentPath, or of an existing name for NewName.

HFSDelete

Description:

***HFSDelete**(ThePath:T): ResultCode:I*

This routine allows you to delete a file or empty folder.

Parameters:

ThePath	The full path to the file or empty folder to be deleted.

Return Value:

ResultCode	Zero if the file or folder was deleted successfully; if an error occurs, the error code will be returned. Usually, such errors will be caused by the specification of a non-existing file or folder, or of a folder that is not empty.

HFSDeleteData

Description:

***HFSDeleteData**(ThePath:T): ResultCode:I*

This routine allows you to delete a file or empty folder.

Parameters:

ThePath .... The full path to the file or empty folder to be deleted.

Return Value:

ResultCode .... Zero if the file or folder was deleted successfully; if an error occurs, the error code will be returned. Usually, such errors will be caused by the specification of a non-existing file or folder, or of a folder that is not empty.

HFSExists

Description:

***HFSExists**(ThePath:T): ResultCode:I*

This function enables you to test for the existence of a file or folder.

Parameters:

ThePath .... A text expression containing the full path to the file or folder to be tested.

Return Value:

ResultCode .... A value of 1 if the entry exists; zero if it does not.

HFSCatToArray

Description:

***HFSCatToArray**(FolderPath:T; ArrayName:S): ResultCode:I*

Creates a text array containing a list of the simple names for all files and subdirectories contained in the specified folder. Folder names will end with a path delimiter (":" on the Macintosh, "\" on Windows), while file names will not.

The only difference between this routine and the new HFSCatToArray2 is the manner in which the second parameter, the array to load with the catalog entry names, is specified. In this routine, a string expression providing the name of the array is used instead of the actual array.

Parameters:

FolderPath .... A text expression containing the full path to the folder whose catalog is to be loaded.

ArrayName .... A string expression containing the name of the text array to be created/ replaced to store the folder's directory. You must declare the array using the ARRAY TEXT statement prior to calling this routine.

Return Value:

ResultCode	Zero if the array was loaded successfully; if an error occurs, the error code will be returned. Usually, such errors will be caused by the specification of a non-existing folder.

HFSCatToArray2

Description:

HFSCatToArray2*(FolderPath:T; Array:X): ResultCode:I*

Creates a text array containing a list of the simple names for all files and subdirectories contained in the specified folder. Folder names will end with a path delimiter (":" on the Macintosh, "\" on Windows), while file names will not.

The only difference between this routine and the original HFSCatToArray is the manner in which the second parameter, the array to load with the catalog entry names, is specified. In this routine, the actual array (or a dereferenced array pointer) is used instead of a string expression providing the name of the array.

Parameters:

FolderPath	A text expression containing the full path to the folder whose catalog is to be loaded.
Array	A text array to be created/replaced to store the folder's directory. You must declare the array using the ARRAY TEXT statement prior to calling this routine.

Return Value:

ResultCode	Zero if the array was loaded successfully; if an error occurs, the error code will be returned. Usually, such errors will be caused by the specification of a non-existing folder.

HFSGetCatCount

Description:

HFSGetCatCount*(FolderPath:T): ResultCode:I*

This routine will return a count of the number of files and folders contained by a specific folder on a mounted volume. Used with HFSGetCatItem, it will enable you to step through the entries in a directory and retrieve the paths to these entries, one at a time.

Parameters:

FolderPath	A text expression containing the full path to the folder whose catalog is to be counted.

Return Value:

ResultCode	The top-level count of files and folders contained in the specified folder. If the specified folder does not exist or an error occurs, a negative error code is returned.

HFSGetCatItem

Description:

***HFSGetCatItem**(FolderPath:T; ItemIndex:I): ItemPath:T*

This routine will enable you to step through the entries in a directory and retrieve the paths to these entries, one at a time. You should use this routine (with HFSGetCatCount) instead of HFSCatToArray whenever the potential number of items is very large, and memory is too tight to create the entire array of entry-names.

Parameters:

FolderPath	A text expression containing the full path to the folder whose catalog is to be counted.
ItemIndex	An integer expression providing the entry-number of the item you wish to retrieve.

Return Value:

ItemPath	The full path name of the specified catalog entry.

GetSystemPath

Description:

***GetSystemPath**: SystemPath:T*

This function returns the fully-qualified path to the system folder on your user's machine.

GetDatabasePath

Description:

***GetDatabasePath**: DatabasePath:T*

This function returns the fully-qualified path to the current database file as a text value.

GetStructPath

Description:

***GetStructPath**: StructurePath:T*

This function returns the path to the user's copy of the 4D structure file as a text value.

GetAppPath

Description:

***GetAppPath**: ApplicationPath:T*

This function returns the fully-qualified path to the user's copy of the 4D application (4th DIMENSION, 4D Runtime, 4D Client or the compiled runtime/application) file, as a text value.

Working With Volumes

FilePack provides these additional utility routines for working directly with volumes:

- The GetDriveVolName routine is used to read the name of a specific volume, such as the user's diskette drive, or to generate a list of all available volumes.
- The GetVolSpace routine allows you to determine the amount of space free on a given volume.
- IsVolLocked will tell you if a given volume can be written to.
- MountVolume is used to procedurally mount a remote shared volume.
- EjectVolume can be used to eject a removable volume, or to dismount a shared volume.

GetDriveVolName

Description:

***GetDriveVolName**(DriveNo:I): VolumeName:S*

Given a physical drive number or a negative volume reference number, this routine returns the volume name. Use this routine for identifying volumes in specific drives, or for generating a list of available volumes.

Parameters:

DriveNo	An integer value specifying either the physical drive containing the volume, or a negative value specifying the logical volume reference number.

Return Value:

VolumeName	The name of the volume in the designated drive, or a null string ("") if the drive is non-existing or empty.

Platform Notes:

On the Macintosh platform, physical drive numbers are generally in the range of 1 to 3 for diskette drives, and greater than 3 for hard disks, CD-ROM drives, etc. Volume reference numbers always range from -1 to -n, where n is the number of mounted volumes. Volume reference numbers are assigned to drives as they are added to the desktop, so you can always assume that drive -1 will be the startup volume.

For Windows, the routines in this package consider the "name" of a volume to be the path to its root directory, i.e. "C:\". Windows does support volume labels, but they are so rarely used, they are worthless for this purpose. The root path is the only consistent naming convention that programmers can follow.

This approach causes a problem, however: removeable volumes for a given drive will all share the same "name" — all diskettes that can be inserted in drive "A:\" will have the same name: "A:\". This will require some existing 4D applications to be reprogrammed, since the logic most people use when asking for a specific diskette is to look for a desired volume name. From now on, we will have to either scan the volume's directory for key files, or trust the user to give us the diskette we have asked for.

The concepts of drive numbers and volume reference numbers as described here have been simulated for Windows as much as possible. Positive drive numbers have a one-to-one correspondence with DOS drive letters, i.e. drive 1 is always "A:\", drive 2 is always "B:\", etc. If you pass a positive drive number to GetDriveVolName for a drive that does not exist, it will return a null string.

Negative drive numbers are, like volume reference numbers on the Mac, relative; passing -2 will return the path of the second available drive. Like on the Mac, the first relative drive (-1) will always be the startup drive.

As an example, say we have a machine with one diskette drive, Windows installed on the C drive, a CD-ROM and two network drives. GetDriveVolName will return the following:

Drive	Volume Name	Volume Ref.	Volume Name
1	"A:\"	-1	"C:\"
2	""	-2	"A:\"
3	"C:\"	-3	"D:\"
4	"D:\"	-4	"E:\"
5	"E:\"	-5	"F:\"
6	"F:\"	-6, etc.	""
7, etc.	""		

This approach should make most existing code work well across platforms without further effort.

GetVolSpace

Description:

GetVolSpace*(VolumeName:S): VolSpace:I*

This function enables you to determine the amount of free space (in bytes) available on a specified volume.

Parameters:

VolumeName .... The name of a mounted volume (such as that returned by GetVolumeName).

Return Value:

VolSpace .... The number of free bytes on the volume specified.

IsVolLocked

Description:

IsVolLocked*(VolumeName:S): ResultCode:I*

Enables you to determine if a volume is write-protected, i.e. if the user has set the sliding tab on the corner of the disk to the write-protected position.

Parameters:

VolumeName .... The name of a mounted volume (such as that returned by GetVolumeName).

Return Value:

ResultCode .... An integer value of one if the specified volume is locked, or zero if it is not.

MountVolume

Description:

MountVolume*(ZoneName:S; ServerName:S; VolumeName:S; UserName:S; PassWord:S): ResultCode:I*

This function enables you to eject or dismount a specified volume.

Parameters:

ZoneName	The name of the desired volume's zone, or "*" if the volume is located in the user's zone.
ServerName	The name of the desired volume's server machine, as specified in the Chooser control panel.
VolumeName	The name of the desired volume.
UserName	The user's name, as specified in the Chooser control panel.
PassWord	The user's password for the specified volume.

Return Value:

ResultCode	Zero if the volume was successfully mounted; if an error occurred, a non-zero error code is returned.

EjectVolume

Description:

EjectVolume*(VolumeName:S): ResultCode:I*

This function enables you to eject or dismount a specified volume.

Parameters:

VolumeName	The name of a mounted volume (such as that returned by GetVolumeName).

Return Value:

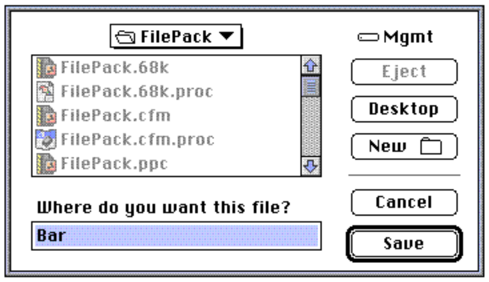
ResultCode	Zero if the volume was successfully ejected; if an error occurred, a non-zero error code is returned. Usually such errors are caused by passing a name of a non-existing or non-ejectable volume.

File System Dialogs

Both the MacOS and Windows provides developers with a set of standardized dialogs that can be used to select file system objects. On the Mac, this facility is known as the Standard File Package, and on Windows, as the Common Dialog Library.

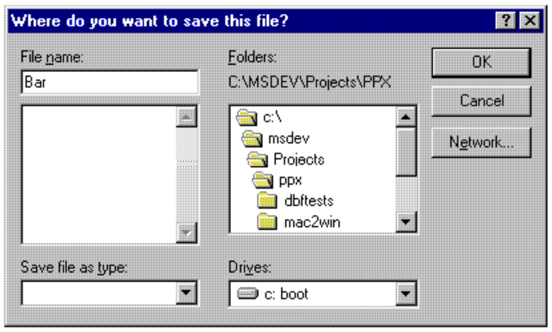
Most applications have the familiar File menu commands called Open… and Save As…. The dialogs displayed by these commands allow the user to navigate through their volumes and folders and either select an existing file to be opened, or specify a name for a new file and the folder to save it in. With the following FilePack routines, your 4D application can make use of these file system dialogs:

- The GetFileName and PutFileName routines allow you to display the file system dialogs used to open and save files.



FilePack.68k
FilePack.68k.proc
FilePack.cfm
FilePack.cfm.proc
FilePack.ppc

Where do you want this file?
Bar



Where do you want to save this file?

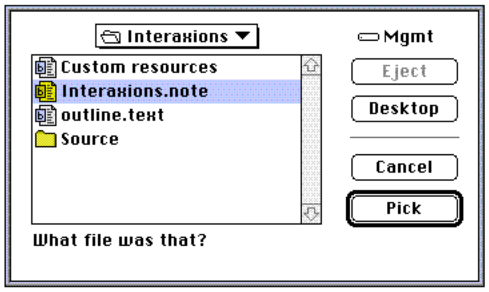
File name: Bar

Folders: C:\MSDEV\Projects\PPX

c:\
msdev
Projects
ppx
dbftests
mac2win

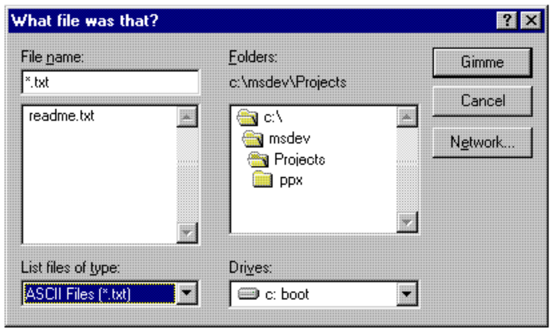
Save file as type:

Drives: c: boot



Custom resources
Interaxions.note
outline.text
Source

What file was that?



What file was that?

File name: *.txt

readme.txt

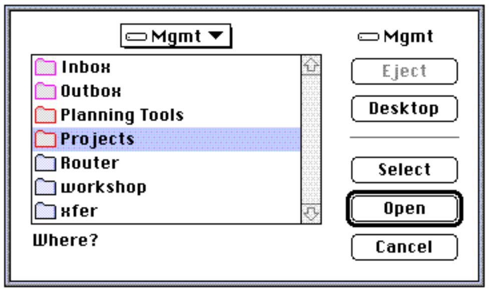
Folders: c:\msdev\Projects

c:\
msdev
Projects
ppx

List files of type: ASCII Files (*.txt)

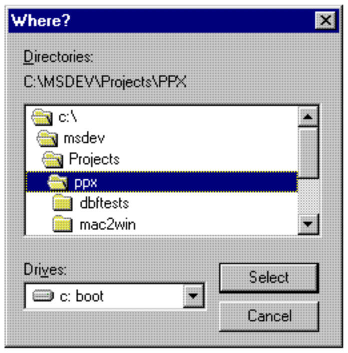
Drives: c: boot

- GetFolderName uses a specially modified version of the standard Open dialog that will allow your users to select a folder, rather than a file.



Inbox
Outbox
Planning Tools
Projects
Router
workshop
xfer

Where?



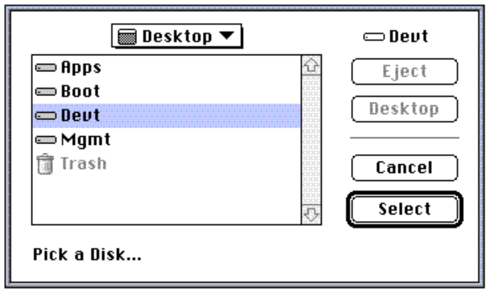
Where?

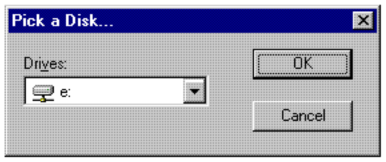
Directories:
C:\MSDEV\Projects\PPX

c:\
msdev
Projects
ppx
dbftests
mac2win

Drives: c: boot

• GetVolumeName and GetDTVolumeName also use customized Open dialogs that allow you to ask the user to select a volume.





The file system automatically maintains a "default" path that each subsequent dialog will display when invoked. It's as though the file system has a "memory," such that once the user navigates to a specific folder using an Open or Save As dialog, that same folder will be displayed when they next invoke such a dialog. There are times, however, when you will want to force the file system to temporarily "forget" where the user went last, and direct it to display a folder of your own choosing. For example, suppose your application maintains a special folder for storing report-definition files for 4D's Quick Report editor; you may wish to direct the file system to display the contents of this special folder before asking the user to select a report definition file, regardless of where the user's last file was selected.

The GetDefaultPath routine will allow you to read the current "default" path, so that you can restore it after your special operation is complete. You can then use the SetDefaultPath routine to set the path to that which you desire for the purpose of the task at hand. The path you specify for this routine will then be immediately visible in a file system dialog, the next time one is displayed.

GetFileName

Description:

***GetFileName**(Prompt:S; Types:S): UserPath:T*

Allows you to display the Standard File Package's Open File dialog, to allow the user of your application to select an existing file, optionally limiting the selectable files to those of specific types.

Parameters:

Prompt	String to be displayed immediately below the list of files and folders in the standard file dialog.
Types	A string expression listing from 1 to 4 4-byte Macintosh file-type codes. Only files with one of the file type codes specified will be visible in the file dialog. If a null string is passed, then all file types will be visible in the dialog.

Return Value:

UserPath	A text value containing the fully-qualified name of the file selected by the user. If the user presses the dialog's Cancel button, a null string is returned.

Platform Notes:

The file types specified are used to build a standard Windows filter list for the dialog, based on 4D's internal file-type mapping table. You can extend this table using the MAP FILE TYPES command.

GetFileName2

Description:

GetFileName2(*Prompt:S; Types:S; Left:I; Top:I; OpenLabel:S): UserPath:T*

Allows you to display the Standard File Package's Open File dialog, to allow the user of your application to select an existing file, optionally limiting the selectable files to those of specific types. The differences between this routine and the original GetFileName are the additional parameters Left and Top for positioning the dialog, and OpenLabel for changing the label of the dialog's Open button.

Parameters:

Prompt	String to be displayed immediately below the list of files and folders in the standard file dialog.
Types	A string expression listing from 1 to 4 4-byte Macintosh file-type codes. Only files with one of the file type codes specified will be visible in the file dialog (and therefore selectable by the user). If a null string is passed, then all file types will be visible in the dialog.
Left	Top .... Integer values specifying the location of the upper-left corner of the dialog. Note that if the position specified places any part of the dialog off the screen, the Standard File Package will ignore these values and place the dialog in its default position.
Top	
OpenLabel	String to be used to label the dialog's Open button. Make sure that the string fits within the button at its usual size—the button will not be resized. Pass an empty string ("") to use the button's default label.

Return Value:

UserPath	A text value containing the fully-qualified name of the file selected by the user. If the user presses the dialog's Cancel button, a null string is returned.

Platform Notes:

The file types specified are used to build a standard Windows filter list, based on 4D's internal file-type mapping table. You can extend this table using the MAP FILE TYPES command.

PutFileName

Description:

PutFileName(*Prompt:S; DefaultName:S): UserPath:T*

Allows you to display the Standard File Package's Save File dialog, to allow the user of your application to specify the name and folder for a new file, optionally allowing you to provide a prompt string and a suggested/default name for the new file.

Parameters:

Prompt	A string to be displayed immediately above the text-edit field for the new file name in the standard file dialog.
DefaultName	A string to be placed in the text-edit area for the new file-name, to act as a default or recommended name for the new file (the user will be able to replace this name with whatever legal name they prefer).

Return Value:

UserPath	The routine returns the fully-qualified name of the file specified by the user. If the user presses the dialog's Cancel button, it will return a null string.

PutFileName2

Description:

***PutFileName2**(Prompt:S; DefaultName:S; Left:I; Top:I): UserPath:T*

Allows you to display the Standard File Package's Save File dialog, to allow the user of your application to specify the name and folder for a new file, optionally allowing you to provide a prompt string and a suggested/default name for the new file. The differences between this routine and the original PutFileName are the additional parameters Left and Top for positioning the dialog.

Parameters:

Prompt	A string to be displayed immediately above the text-edit field for the new file name in the standard file dialog.
DefaultName	A string to be placed in the text-edit area for the new file-name, to act as a default or recommended name for the new file (the user will be able to replace this name with whatever legal name they prefer).
Left	Top .... Integer values specifying the location of the upper-left corner of the dialog. Note that if the position specified places any part of the dialog off the screen, the Standard File Package will ignore these values and place the dialog in its default position.
Top	

Return Value:

UserPath	The routine returns the fully-qualified name of the file specified by the user. If the user presses the dialog's Cancel button, it will return a null string.

GetFolderName

Description:

***GetFolderName**(Prompt:S): UserPath:T*

Presents a modified standard-file dialog, allowing the user to select an existing folder, and returns the fully-qualified pathname for the selected folder. The dialog has an additional button called Select that will be enabled whenever a folder is highlighted in the file list area. The Select button allows the user to close the dialog and return the fully-qualified pathname of the highlighted folder to your procedure.

Parameters:

Prompt	A string to be displayed immediately below the list of folders in the standard file dialog.

Return Value:

UserPath	The fully-qualified name of the folder selected by the user. The text value will end with a colon. If the user presses the dialog's Cancel button, returns a null string.

GetFolderName2

Description:

GetFolderName2*(Prompt:S; Left:I; Top:I): UserPath:T*

Presents a modified standard-file dialog, allowing the user to select an existing folder, and returns the fully-qualified pathname for the selected folder. The dialog has an additional button called Select that will be enabled whenever a folder is highlighted in the file list area. The Select button allows the user to close the dialog and return the fully-qualified pathname of the highlighted folder to your procedure. The differences between this routine and the original GetFolderName are the additional parameters Left and Top for positioning the dialog.

Parameters:

Prompt	A string to be displayed immediately below the list of folders in the standard file dialog.
Left	Top .... Integer values specifying the location of the upper-left corner of the dialog. Note that if the position specified places any part of the dialog off the screen, the Standard File Package will ignore these values and place the dialog in its default position.
Top	

Return Value:

UserPath	The fully-qualified name of the folder selected by the user. The text value will end with a colon. If the user presses the dialog's Cancel button, returns a null string.

GetVolumeName

Description:

GetVolumeName*(Prompt:S): VolumeName:S*

This function allows you to provide the user with a means of selecting a volume, and will return the selected volume's name. On the Macintosh, it displays a customized Standard-File dialog showing nothing but the volume name, the Eject, Drive, Select and Cancel buttons, and a prompt that you supply. The user can use the Drive and Eject buttons to display the name of the volume they wish to select, and then click the Select or Cancel button to dismiss the dialog.

Note: This routine may not work with some INITs that extend the Standard File Package under System 7. If you have these problems, you should use GetDTVolumeName.

Parameters:

Prompt	A string expression that will display along the bottom of the dialog.

Return Value:

VolumeName	The name of the selected volume, or a null string ("") if the user cancels.

GetVolumeName2

Description:

GetVolumeName2*(Prompt:S; Left:I; Top:I): VolumeName:S*

This function allows you to provide the user with a means of selecting a volume, and will return the selected volume's name.

On the Macintosh, it displays a customized Standard-File dialog showing nothing but the volume name, the Eject, Drive, Select and Cancel buttons, and a prompt that you supply. The user can use the Drive and Eject buttons to display the name of the volume they wish to select, and then click the Select or Cancel button to dismiss the dialog. The differences between this routine and the original GetVolumeName are the additional parameters Left and Top for positioning the dialog.

Parameters:

Prompt	A string expression that will display along the bottom of the dialog.
Left	Top .... Integer values specifying the location of the upper-left corner of the dialog. Note that if the position specified places any part of the dialog off the screen, the Standard File Package will ignore these values and place the dialog in its default position.
Top	

Return Value:

VolumeName	The name of the selected volume, or a null string ("") if the user cancels.

GetDTVolumeName

Description:

GetDTVolumeName*(Prompt:S): VolumeName:S*

This function allows you to provide the user with a means of selecting a volume, and will return the selected volume's name. When called under System 6.0, it behaves exactly the same as the GetVolumeName routine. When called under System 7.0 (or some future release of 6.0 that supports the new Standard File Package and Aliases), it will display the familiar standard file dialog, forcing the user to view the Desktop pseudo-directory and to select a volume from its scrollable list. The advantage of this approach under System 7 is that you can place aliases to unmounted shared/server volumes on the desktop, and the user will be able to select and mount these volumes. Another advantage is that this routine is much less likely to run into problems with INITs that patch or extend the Standard File Package.

Parameters:

Prompt	A string expression that will display along the bottom of the dialog.

Return Value:

VolumeName	The name of the selected volume, or a null string ("") if the user cancels.

GetDTVolName2

Description:

***GetDTVolName2**(Prompt:S; Left:I; Top:I): VolumeName:S*

This function allows you to provide the user with a means of selecting a volume, and will return the selected volume's name. When called under System 6.0, it behaves exactly the same as the GetVolumeName routine. When called under System 7.0 (or some future release of 6.0 that supports the new Standard File Package and Aliases), it will display the familiar standard file dialog, forcing the user to view the Desktop pseudo-directory and to select a volume from its scrollable list. The differences between this routine and the original GetDTVolumeName are the additional parameters Left and Top for positioning the dialog.

Parameters:

Prompt	A string expression that will display along the bottom of the dialog.
Left	Top .... Integer values specifying the location of the upper-left corner of the dialog. Note that if the position specified places any part of the dialog off the screen, the Standard File Package will ignore these values and place the dialog in its default position.
Top	

Return Value:

VolumeName	The name of the selected volume, or a null string ("") if the user cancels.

GetDefaultPath

Description:

GetDefaultPath

Determines the path name for the standard-file package's current "default" directory. The "default" directory is the one most recently viewed by the user in a standard-file package dialog, and is the one the user will expect to see the next time a standard-file package dialog is displayed.

Return Value:

DefaultPath	A text value containing the fully-qualified path name for the directory most recently seen by the user in a standard-file package dialog.

SetDefaultPath

Description:

***SetDefaultPath**(NewDefPath:T): ResultCode:I*

Allows you to specify a new folder for the standard-file package's current "default" directory. The "default" directory is the one most recently viewed by the user in a standard-file package dialog, and is the one the user will expect to see the next time a standard-file package dialog is displayed.

Parameters:

NewDefPath	The fully-qualified path of the directory you wish your user to see the next time any standard-file package dialog is displayed. The pathname must specify an existing directory (folder or volume).

Return Value:

ResultCode	Zero if the directory was assigned successfully; if an error occurs, the error code will be returned. Usually, such errors will be caused by the specification of a directory that does not exist.